A Massively Parallel Algorithm for the Approximate Calculation of Inverse p-th Roots of Large Sparse Matrices

Michael Lass
Department of Computer Science, Paderborn University
Paderborn, Germany
michael.lass@uni-paderborn.de

Stephan Mohr
Barcelona Supercomputing Center
Barcelona, Spain
stephan.mohr@bsc.es

Hendrik Wiebeler
Department of Chemistry, Paderborn University
Paderborn, Germany
wie@mail.uni-paderborn.de

Thomas D. Kühne
Department of Chemistry, Paderborn University
Paderborn, Germany
tdkuehne@mail.uni-paderborn.de

Christian Plessl
Department of Computer Science, Paderborn University
Paderborn, Germany
christian.plessl@uni-paderborn.de

ABSTRACT

We present the *submatrix method*, a highly parallelizable method for the approximate calculation of inverse p-th roots of large sparse symmetric matrices which are required in different scientific applications. Following the idea of Approximate Computing, we allow imprecision in the final result in order to utilize the sparsity of the input matrix and to allow massively parallel execution. For an $n \times n$ matrix, the proposed algorithm allows to distribute the calculations over n nodes with only little communication overhead. The result matrix exhibits the same sparsity pattern as the input matrix, allowing for efficient reuse of allocated data structures.

We evaluate the algorithm with respect to the error that it introduces into calculated results, as well as its performance and scalability. We demonstrate that the error is relatively limited for well-conditioned matrices and that results are still valuable for error-resilient applications like preconditioning even for ill-conditioned matrices. We discuss the execution time and scaling of the algorithm on a theoretical level and present a distributed implementation of the algorithm using MPI and OpenMP. We demonstrate the scalability of this implementation by running it on a high-performance compute cluster comprised of 1024 CPU cores, showing a speedup of 665× compared to single-threaded execution.

CCS CONCEPTS

• **Mathematics of computing → Computations on matrices**; • **Theory of computation → Numeric approximation algorithms**; **Massively parallel algorithms**; **Distributed algorithms**; • **Computing methodologies → Massively parallel algorithms**; **Distributed algorithms**; • **Applied computing →** *Chemistry*; *Physics*;

KEYWORDS

approximate computing, linear algebra, matrix inversion, matrix p-th roots, numeric algorithm, parallel computing

ACM Reference Format:
Michael Lass, Stephan Mohr, Hendrik Wiebeler, Thomas D. Kühne, and Christian Plessl. 2018. A Massively Parallel Algorithm for the Approximate Calculation of Inverse p-th Roots of Large Sparse Matrices. In *PASC '18: Platform for Advanced Scientific Computing Conference, July 2–4, 2018, Basel, Switzerland*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3218176.3218231

PASC '18, July 2–4, 2018, Basel, Switzerland

ACM ISBN 978-1-4503-5891-0/18/07…$15.00
https://doi.org/10.1145/3218176.3218231

1 INTRODUCTION

Inverse matrices and inverse p-th roots, i.e., $A^{-1/p}$ for a given matrix A, are important for various applications in the area of scientific computing. Examples are preconditioning, solving systems of linear equations and linear least squares problems, non-linear optimization, Kalman filtering and solving generalized eigenvalue problems, in particular to solve Schrödinger and Maxwell equations.

In many of these applications, the involved matrices are very large, containing billions of entries, and also sparse. For increasing problem sizes, it becomes more and more important to exploit this sparsity to save both computational effort and memory resources. However, the inverse and inverse p-th roots of a sparse matrix are typically not sparse anymore, which makes exploitation of the sparsity difficult. The novel approach proposed in this work is to only compute an approximate solution for the inverse p-th roots of large matrices and enforcing that the result has the same sparsity pattern as the input matrix. At the same time the method allows massive parallelization of the required computations. Hence, the proposed method adopts the idea of *Approximate Computing*, which denotes the concept of sacrificing accuracy of computation results in order to increase the efficiency of these computations [15]. We call the method proposed in this work the *submatrix method*.

There are multiple applications where approximate solutions for inverse p-th roots are of use. One of these applications is preconditioning, where efficient computations can be more important than accuracy. Another particularly important application area are electronic structure methods to approximately solve the electronic Schrödinger equation [16]. Interestingly it has been shown that, for

large system sizes, the representing matrices become eventually all sparse [10]. Underlying this is the concept of "nearsightedness of electronic matter", which states that, at fixed chemical potential, the electronic density depends just locally on the external potential [27]. Consequently, with increasing system size, the number of nonzero elements in matrices used to simulate these systems only increases linearly, i.e., the density decreases also linearly as the system size increases. Specialized solver libraries such as CheSS [25] exploit this behavior but still require processing large sparse matrices. The method proposed in this work is especially suitable in these applications since it exploits the increasing sparsity of the matrices and allows scaling the parallelism with the matrix size. Since linear-scaling electronic structure methods entail approximation in practice [28], it can be sufficient to calculate an approximation for the inverse p-th root of the involved matrices.

The remainder of this work is structured as follows. In Section 2 we give a brief overview of related work in the field of matrix inversion, calculation of p-th roots and parallelization of these operations. In Section 3 we present the submatrix method in detail. Afterwards, we discuss the error that is introduced by using the submatrix method in Section 4 and its complexity and scalability in Section 5. Finally, we present an implementation of the proposed method using MPI [24] and OpenMP [7] in Section 6 and use it to evaluate the scalability of the method in practice using a large compute cluster. We conclude in Section 7.

2 FOUNDATIONS AND RELATED WORK

Due to the importance of matrix inversions and inverse p-th roots and the computational complexity of these operations, there is a large variety of ressources on different numerical methods and on efficient implementation of these methods. We want to give a brief overview of commonly used algorithms, ready-to-use implementations and related work on parallelization of the required calculations. However, an exhaustive coverage of available methods is outside the scope of this work.

The inverse X of a matrix A fulfils the equation

$$AX = I, \tag{1}$$

where I is the identity matrix. X can therefore be determined by solving this equation using Gaussian elimination or by calculating and using an LU or LUP decomposition of A. For symmetric matrices, one can compute the singular value decomposition (SVD), and invert all singular values. A different approach is to reduce the problem of determining the inverse of an $n \times n$ input matrix to calculating the inverse of two $n/2 \times n/2$ matrices and performing several matrix-matrix multiplications. Following this idea recursively, inverting a matrix can be reduced to performing only matrix-matrix multiplications [6, Ch. 28.2]. Lastly, iterative methods can be used to find the inverse of a matrix, such as the Newton-Schulz iteration scheme [30].

In literature, several approaches can be found to parallelize matrix inversion or calculation of LU and SV decompositions. For example, Van der Stappen et al. [32] present an algorithm for parallel calculation of the LU decomposition on a mesh network of transputers where each processor holds a part of the matrix. Shen [31] evaluates techniques for LU decomposition distributed over nodes that are connected via slow message passing. Dongarra et al. [9] demonstrate an optimized implementation of matrix inversion on a single multicore node, focusing on the minimization of synchronization between the different processing cores. There are also algorithms specialized on specific applications, such as the one described by Lin et al. [22] which can be used in 2D electronic structure calculations to only calculate selected parts of the inverse of a sparse matrix. For parallel calculation of the SVD, Berry et al. [3] provide an extensive overview of parallelizable methods.

For the calculation of the p-th root $A^{1/p}$ of a matrix A, a commonly used algorithm is the one described by Higham and Lin [11, 12]. For the calculation of inverse p-th roots, i.e., $A^{-1/p}$, there are also iterative methods available, such as the ones described by Bini et al. [4] and Richters et al. [29], which reduce the problem to repeated matrix-matrix multiplications. For symmetric matrices, the solution can again also be computed by building the SVD of the input matrix and applying the operation of interest to all singular values of the matrix. For very large sparse matrices, calculation of the SVD is typically avoided. Instead methods are used where only the largest eigenvalues of the matrix are calculated. Iterative methods to do so are the Lanczos algorithm [18] and the Arnoldi iteration [2].

Implementations for the calculation of the LU and SV decompositions and matrix inversion are part of LAPACK [1], a popular software library for numerical linear algebra. For solving large sparse systems and calculation of singular values, ARPACK [21] is a well known library which is based on the Arnoldi iteration. There exist different implementations of these libraries, as well as bindings for many different programming languages. With ScaLAPACK [5] and P_ARPACK [23], there exist extensions of these libraries targeting parallel execution on distributed memory systems using MPI for message passing.

3 ALGORITHM DESCRIPTION

The fundamental concept of the proposed submatrix method is to divide the large, sparse input matrix into several smaller and dense submatrices and to apply the desired operation, such as inversion or calculation of inverse p-th roots, to all of these submatrices instead of the original matrix. Afterwards the results of these operations are used to construct an approximate result matrix. The overall procedure is shown in Figure 1. In the following, we describe the main steps of this method, in particular, building the submatrices and assembling the final result matrix, in detail. Note that, although we only discuss a column-based approach for building the submatrices, the method can as well be applied in a row-based manner since we are dealing with symmetric matrices. For ease of reading, we will often only mention the inversion of matrices in the remainder of this section, while the same also holds for the calculation of inverse p-th roots.

3.1 Building the submatrices

The process of building the submatrices is shown in Algorithm 1. To simplify comprehension of the algorithm, all matrices have a dense representation in our pseudocode. As will be discussed in Section 3.4, in practice a sparse representation should be used for the sparse input and output matrices.

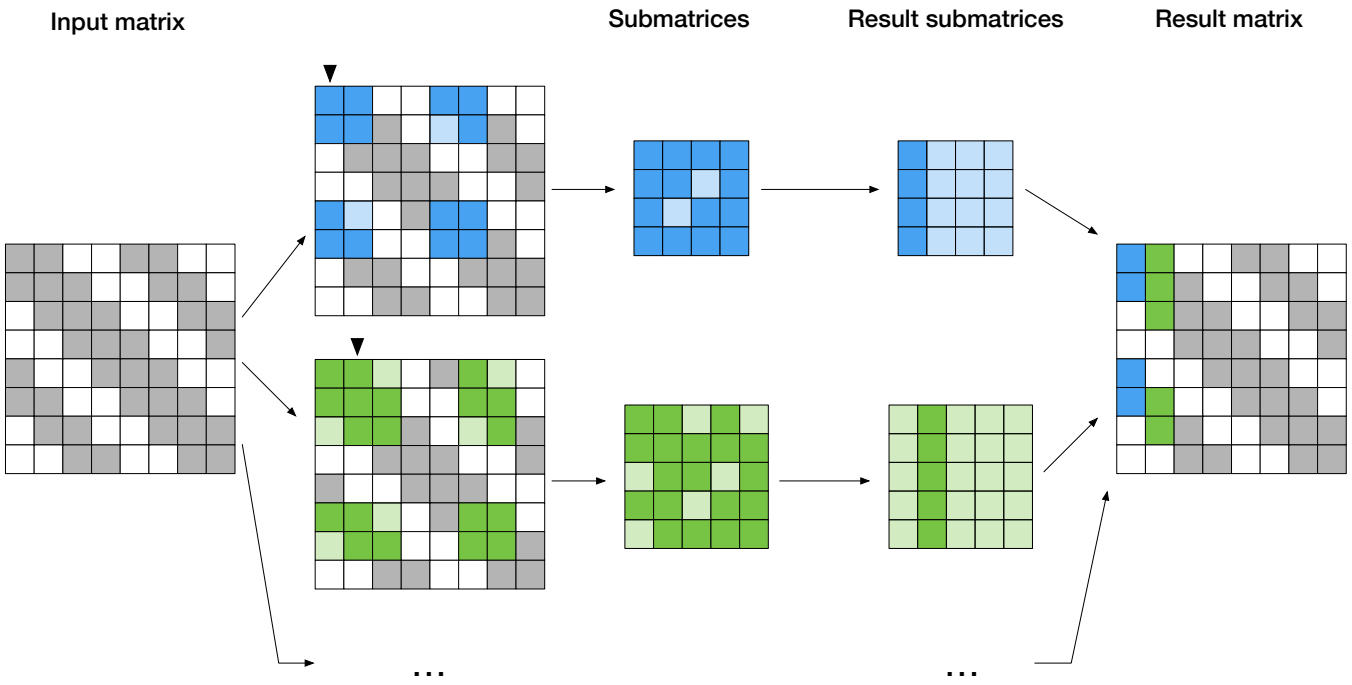

Figure 1: Overview of the submatrix method

To construct the j-th submatrix, the j-th column of the input matrix A is evaluated. We determine the set R of row indices i for which $A_{i,j} \neq 0$. The submatrix is then constructed by taking all values $A_{x,y}$ from the input matrix where $x, y \in R$. For an input matrix of size $n \times n$ we obtain a set of n submatrices. The size of each submatrix is determined by the number of nonzero elements in the corresponding column of the input matrix.

Algorithm 1 Construction of submatrices

```
n ← number of rows/columns of input matrix
A[1...n][1...n] ← input matrix
for j ← 1...n do
    R ← ∅
    for i ← 1...n do
        if A[i][j] ≠ 0 then
            R ← R ∪ {i}
        end if
    end for
    m ← R.length()
    for k ← 1...m do
        for l ← 1...m do
            submatrices[j][k][l] ← A[R[k]][R[l]]
        end for
    end for
    indices[j] ← R          ▷ required later on for result assembly
end for
```

3.2 Performing submatrix operations

For all of the submatrices, we now perform the operation which should originally be performed on the input matrix, i.e., we either invert all submatrices or calculate their inverse p-th roots. Note that the method and implementation for these submatrix operations can be freely selected and this choice is entirely orthogonal to the submatrix method described in this work.

3.3 Assembling the result matrix

After having applied the matrix operation of interest to each submatrix, we have n result submatrices. From these result submatrices we assemble an approximate solution X for the whole matrix. This procedure is shown in Algorithm 2. Similar to the construction of the submatrices, the j-th column of the final result matrix is determined by the j-th result submatrix. We take the values from the column of the result submatrix which was originally filled with values from the j-th column of the input matrix, and copy them back to their original position in the original matrix.

3.4 Implementation notes

Although the inverse of a sparse matrix is typically not sparse, the approximate solution provided by the submatrix method exhibits the exact same sparsity pattern as the input matrix. This allows for efficient implementation of the method based on matrices in the compressed sparse column (CSC) format which consists of a value array (`val`), a list of row indices (`row_ind`) and a list of column pointers (`col_ptr`). In particular, the result assembly stage can be implemented by concatenating the corresponding columns of all result submatrices to obtain the value array `val` for the approximate

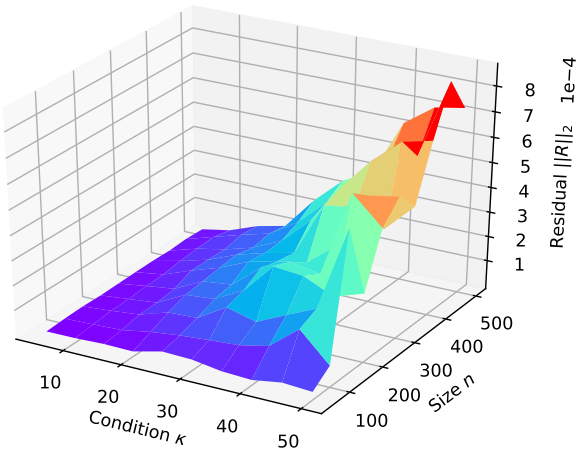

Figure 2: Residual for approximately calculated inverse of random matrices using submatrix method, for different sizes and condition numbers.

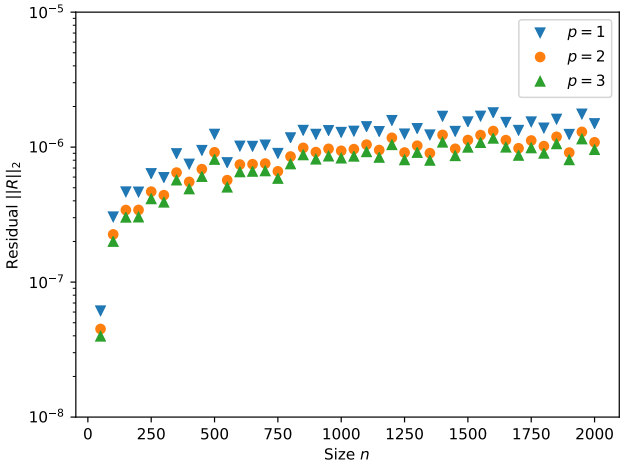

Figure 3: Residual for approximately calculated inverse of random matrices with $\kappa = 2$ using submatrix method, in relation to size of input matrix.

result matrix. `row_ind` and `col_ptr` from the input matrix can be reused for the output matrix without any changes. If the method is applied in a row-based manner, the same holds for matrices in the compressed sparse row (CSR) format.

4 APPLICABILITY AND APPROXIMATION ERROR

Evidently, the result obtained by the submatrix method is only an approximation of the correct result. Whether this result is still of use for an application depends on three aspects: Is the application able to deal with results that contain a certain error, how large can this error be to be still tolerable and how large is the error introduced into results by using the submatrix method. In this section, we first characterize the error based on random sparse matrices and afterwards apply the submatrix method to two application scenarios and show that using the submatrix method to process real-world matrices in these application yields good results. Finally we discuss means to influence the approximation error.

Algorithm 2 Assembly of result matrix

```
n ← number of rows/columns of input matrix
indices[1 ... n] ← from submatrix generation stage
submatrices[1 ... n][1 ... ?][1 ... ?] ← result matrices
X ← zeros(n × n)
for j ← 1 ... n do
    R ← indices[j]
    m ← R.length()
    for i ← 1 ... m do
        X[R[i]][j] ← submatrices[j][i][R.indexof(j)]
    end for
end for
```

4.1 Error for random input matrices

To get an impression about the error introduced for arbitrary symmetric positive-definite matrices, we generate random matrices A using the `sprandsym`[1] function in Matlab. This allows us to sweep over different sizes n, densities d and condition numbers κ and assess the influence of these matrix properties onto the eror. For each set of these parameters, we generate ten different matrices. For all of these matrices we then use the submatrix method to obtain an approximate solution X for the inverse p-th root $A^{-1/p}$. To assess the error of these results, we calculate the spectral norm of the residuals

$$\|R\|_2 = \left\|X^p A - I\right\|_2 . \tag{2}$$

Since for a precise solution it should hold that $X^p A = I$, R is a good indicator for the introduced error. We choose the spectral norm of R as a metric because in contrast to other matrix norms like the Frobenius norm it is relatively invariant of the matrix size. The spectral norm of a matrix is defined as

$$\|M\|_2 = \sqrt{\lambda_{\max}(M^H M)}, \tag{3}$$

where $\lambda_{\max}(M)$ denotes the largest eigenvalue of M and M^H is the Hermitian transpose of M.

Our initial evaluation has not shown a significant influence of the density of the randomly generated matrices onto the precision of the result. We therefore neglect this parameter in the evaluation presented here, focus on matrices with density $d = 0.05$ and discuss the influence of the size and the condition number of the matrices. Figure 2 shows the relationship between these matrix properties and the calculated residual for $p = 1$. It shows that the error increases for matrices with larger size and larger condition numbers. For small matrices, the error stays relatively low even for higher condition numbers. Similarly, for well-conditioned matrices, the error stays low even for large matrices.

[1] `sprandsym(size,density,1/condition,kind)` with `kind=1`

To demonstrate the latter, we now focus on well-conditioned matrices with $\kappa = 2$ and $d = 0.05$, varying only their size. Results are shown in Figure 3. It shows that for a fixed condition number, the error introduced by using the submatrix method is limited even when further increasing the matrix size. As shown, this not only holds for calculating the inverse of a matrix but also for calculating inverse p-th roots where $p > 1$.

4.2 Applicability to other matrix operations

Throughout this work we describe the submatrix method as a method to calculate inverse p-th roots of matrices. In fact, the method can be applied to similar matrix operations as well, such as the calculation of positive p-th roots where the error behaves very similar as discussed in Section 4.1. In contrast to the case of inverse p-th roots, the residual is calculated as $R = X^p A^{-1} - I$, which requires a matrix inversion itself. Hence, the residual cannot be calculated efficiently at runtime to assess the error of an approximately computed result. Inverse p-th roots are therefore, and due to their variety of target applications, presented as the main target in this work.

4.3 Application within electronic structure codes

In Section 4.1 we demonstrated that using the submatrix method for well-conditioned matrices yields results very similar to a precisely calculated solution. However, whether errors are acceptable in an application depends on the the kind of matrices used in the application and the effect that small deviations have on the final result. In this section, we show a specific application of the submatrix method and demonstrate its limited influence on the final result.

We choose one specific time-consuming kernel of effective single-particle theory, the orthogonalization of a set of n non-orthogonal basis functions, and assess the impact of using the submatrix method within this kernel on the band-structure energy E_{BS}, which corresponds to the sum of eigenvalues of the Hamilton matrix H. For that purpose, we first compute the so-called overlap matrix $S \in \mathbb{R}^{n \times n}$, where $S_{i,j} = \langle \varphi_i | \varphi_j \rangle$ and φ_i are the n non-orthogonal basis functions spanning the Hilbert space. Specifically, we consider overlap matrices of bulk liquid water systems for different numbers of molecules and hence different dimension of the the overlap matrix, as computed using a Daubechies Wavelet-based density functional theory (DFT) code [26].

Table 1: Influence of using the submatrix method for orthogonalization of basis functions onto electronic band structure energy calculations.

Matrix size n	E_{BS}	$E_{\mathrm{BS}}^{\mathrm{sm}}$	ΔE_{rel}
768	-372.83597	-372.83600	6.96×10^{-8}
1536	-747.13928	-747.13933	7.60×10^{-8}
3072	-1492.25282	-1492.25297	1.01×10^{-7}
6144	-2986.25656	-2986.25683	8.76×10^{-8}
12288	-5976.31525	-5976.31576	8.64×10^{-8}
24576	-11951.19504	-11951.19598	7.85×10^{-8}

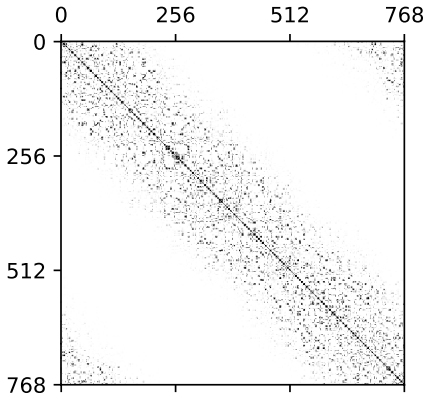

Figure 4: Structure of an examined overlap matrix.

Figure 4 shows one such overlap matrix for a system of 128 H_2O molecules. The matrix has a banded structure and in this case a density of $d = 0.25$ and a condition number of around $\kappa = 1.5$. For increasing system sizes, the density decreases lineraly with n, for example $d = 0.12$ for $n = 1536$, $d = 0.06$ for $n = 3072$ etc., while condition numbers remain approximately constant at around $\kappa = 1.5$.

In addition to the overlap matrices S, we also extract the density matrix P as well as the Hamilton matrix H from our Wavelet-based DFT code. This allows us to calculate the band-structure energy as

$$E_{\mathrm{BS}} = \mathrm{tr}(PH). \tag{4}$$

To orthogonalize the Hamilton matrix, we calculate

$$H_{\mathrm{ortho}} = HS^{-1}, \tag{5}$$

where S^{-1} is computed using the submatrix method. From this, we again calculate the band-structure energy as

$$E_{\mathrm{BS}}^{\mathrm{sm}} = \mathrm{tr}(SPH_{\mathrm{ortho}}), \tag{6}$$

and evaluate the relative error caused by the approximative inversion of S as

$$\Delta E_{\mathrm{rel}} = \left| \frac{E_{\mathrm{BS}} - E_{\mathrm{BS}}^{\mathrm{sm}}}{E_{\mathrm{BS}}} \right|. \tag{7}$$

Results are shown in Table 1. For all evaluated matrix sizes, the relative error caused by using the submatrix method for orthogonalization is rather small and throughout below or around 10^{-7}.

The band structure of the considered overlap matrices may suggest that the low deviation between E_{BS} and $E_{\mathrm{BS}}^{\mathrm{sm}}$ comes from the similarity between S and the identity matrix I, and therefore I could be used in Equation (5) as an approximation for S^{-1}. However, doing so leads to relative errors between 1.15×10^{-2} and 1.21×10^{-2}, i.e., five orders of magnite higher than when using the approximate inverse provided by the submatrix method.

4.4 Application as preconditioner

In the last section we showed an application where the input matrices are well conditioned and the results provided by the submatrix method are good enough for the application. Now we demonstrate

using the submatrix method to process ill-conditioned matrices in order to obtain a preconditioner that can be used to iteratively solve systems of linear equations.

To demonstrate this application scenario, we obtain sparse, symmetric, positive definite matrices from the SuiteSparse matrix library [8]. We select all matrices A with size $1000 \leq n \leq 5000$ that fulfill these requirements. For these matrices we solve the system

$$Ax = b, \quad b = [1, 1, \ldots, 1]^T \tag{8}$$

using the Conjugate Gradient (CG) method. We set the threshold for the residual to 10^{-6} and limit the number of iterations by $2n$. Table 2 shows the number of iterations required for CG to converge towards a solution. For preconditioning, we use the submatrix method to obtain an approximate solution for

$$K \approx A^{-1/2}. \tag{9}$$

Instead of solving Equation (8), we now solve the system given by

$$K^T AKy = K^T b \tag{10}$$

using the CG method. The solution x for Equation (8) can then be computed as

$$x = Ky. \tag{11}$$

Again, results are shown in Table 2. For comparison, we also include the number of iterations required when using an ILU(0)[2] preconditioner. The results show that using the submatrix method for preconditioning is not only competitive to the use of ILU(0) but enables CG to converge in more of the cases.

Table 2: Number of iterations required to solve Equation (8) for different matrices A using CG with different preconditioners.

Matrix	n	κ	None	SM	ILU(0)
1138_bus	1138	8.5×10^{06}	2120	151	139
bcsstk08	1074	2.6×10^{07}	—	41	27
bcsstk09	1083	9.5×10^{03}	194	56	—
bcsstk10	1086	5.2×10^{05}	—	85	182
bcsstk11	1473	2.2×10^{08}	—	273	477
bcsstk12	1473	2.2×10^{08}	—	273	477
bcsstk13	2003	1.1×10^{10}	—	409	—
bcsstk14	1806	1.2×10^{10}	—	54	262
bcsstk15	3948	6.5×10^{09}	—	177	591
bcsstk16	4884	4.9×10^{09}	464	32	35
bcsstk21	3600	1.8×10^{07}	—	224	—
bcsstk23	3134	2.7×10^{12}	—	1269	—
bcsstk24	3562	2.0×10^{11}	—	300	244
bcsstk26	1922	1.7×10^{08}	—	325	337
bcsstk27	1224	2.4×10^{04}	907	66	19
bcsstk28	4410	9.5×10^{08}	—	668	755
bcsstm12	1473	6.3×10^{05}	2790	7	12
Chem97ZtZ	2541	2.5×10^{02}	86	10	1
crystm01	4875	2.3×10^{02}	70	8	2
ex10hs	2548	5.5×10^{11}	—	—	—
ex10	2410	9.1×10^{11}	—	—	—
ex13	2568	1.1×10^{15}	—	—	—
ex33	1733	7.0×10^{12}	—	1052	—
ex3	1821	1.7×10^{10}	—	—	—
ex9	3363	1.2×10^{13}	—	—	—
mhd3200b	3200	1.6×10^{13}	—	6	3
mhd4800b	4800	8.2×10^{13}	—	6	2
msc01050	1050	4.6×10^{15}	—	—	—
msc01440	1440	3.3×10^{06}	—	89	155
msc04515	4515	2.3×10^{06}	4411	357	—
nasa1824	1824	1.9×10^{06}	—	275	264
nasa2146	2146	1.7×10^{03}	282	67	12
nasa2910	2910	6.0×10^{06}	—	282	760
nasa4704	4704	4.2×10^{07}	—	1100	570
plat1919	1919	1.2×10^{17}	—	—	—
plbuckle	1282	1.3×10^{06}	1965	76	69
sts4098	4098	2.2×10^{08}	—	67	119
Trefethen_2000	2000	1.6×10^{04}	435	6	5

4.5 Controlling the approximation error

We have demonstrated the use of the submatrix method in two applications that can highly benefit from the speedup and the additional parallelism and still yields good results. However, there may be applications that are less tolerant to errors but still can benefit from using the submatrix method.

If an application requires a lower error than what is provided by the solution calculated using the submatrix method, an iterative method like those described by Bini et al. [4] and Richters et al. [29] can be used to refine the solution obtained from the submatrix method. The result obtained by using the submatrix method then acts as an initial guess for these iterative methods. While we validated that such a refinement of a solution generated by the submatrix method works in principle and converges within very few iterations, a detailed evaluation of combining the submatrix method with iterative methods remains for future work.

In the contrary case, if the application has a particularly high resiliency against errors in the inverse matrix, the submatrix method can also be combined with other approximation techniques to achieve further performance gains. Since using the submatrix method is orthogonal to the implementation of the operations performed on the single submatrices, these submatrix calculations can be performed in an approximate manner as well. Using an iterative method, precision can be scaled by the number of iterations. Additionally, calculations can be performed using low precision arithmetic or other Approximate Computing techniques [19].

5 COMPLEXITY AND SCALIBILITY

We now want to discuss the time complexity and scalability of the submatrix method and show that, although for an $n \times n$ matrix n submatrices need to be processed, it can still provide a significant reduction in time required for determining a matrix inverse or its inverse p-th root.

Note that the considerations in this section only hold if the density of the input matrix shows in each of its columns (or rows). An obvious counterexample are arrowhead matrices, where using

[2] incomplete LU decomposition with zero fill-in

the submatrix method cannot provide any speedup since the first submatrix has the same size as the original input matrix.

5.1 Single-threaded scenario

We first want to discuss the general complexity of matrix inversion, both using conventional methods and using our proposed submatrix method. While from a theoretical standpoint, inversion of matrices is not harder than multiplication, and therefore $O(n^{2.81})$ using Strassen's algorithm [6, pp. 79, 829], or even $O(n^{2.373})$ using Coppersmith and Winograd's algorithm [20], in practice methods such as Gaussian elimination or building and using the LU decomposition for inversion which have time complexity $O(n^3)$ are commonly used. In the following, we define $I(n)$ as the time required for a precise matrix inversion, abstracting from a concrete implementation.

For a sparse $n \times n$ input matrix, using the submatrix method requires performing n matrix inversions for smaller but dense matrices. To be more efficient in a single-threaded application scenario, these submatrices have to be significantly smaller than the original input matrix. In the following, we assume a uniformly filled, sparse input matrix. Let d be the density of this matrix, then the average size $m \times m$ of the submatrices is determined by:

$$m = d \cdot n. \tag{12}$$

If the density d is small enough, such that

$$n \cdot I(d \cdot n) < I(n), \tag{13}$$

then the submatrix method has lower run time than a precise inversion, even in a single-threaded scenario.

We now want to determine, by what rate the density d has to decrease so that for increasing matrix sizes the asymptotic run time does not grow faster than using conventional methods for matrix inversion. We therefore assume that matrix inversion has at least time complexity n^2, i.e., $I(n) = \Omega(n^2)$. To fulfill Equation (13), it then needs to hold that for sufficiently large n

$$\begin{aligned} n \cdot (d \cdot n)^2 &< n^2 \\ d &< n^{-0.5}. \end{aligned} \tag{14}$$

From this we can deduce the following asymptotic relation:

$$d = O(n^{-0.5}) \Rightarrow S(n, d) = O(I(n)), \tag{15}$$

where $S(n, d)$ is the time required to calculate an approximate inverse using the submatrix method. If d decreases faster than with rate $n^{-0.5}$, then $S(n, d)$ increases slower than $I(n)$ for larger n. Using methods where $I(n) = \Omega(n^3)$ further relaxes the requirements on d, in that $d = O(n^{-1/3})$ suffices to fulfill Equation (13) for large n.

Note that we neglect the time required for building the submatrices and assembling the final result matrix, as their influence on execution time is negligible compared to the involved matrix inversions in asymptotic considerations.

5.2 Parallel execution of submatrix operations

Although using the submatrix method can reduce execution time even in a single-threaded environment for very sparse matrices, its strength is to allow massively parallel execution. All submatrix operations are entirely independent from each other such that the inversion of an $n \times n$ matrix can be distributed over n compute nodes. Each compute node can construct its own submatrix from the input matrix. The final result matrix has to be assembled on a single node but as described in Section 3.4, this step consists of a simple concatenation of n arrays. Communication between nodes is only required for initial data distribution and for the final collection of all results. Provided that n compute nodes can be used for execution of the algorithm, a speedup is already achievable if all submatrices are significantly smaller than the original input matrix, i.e., each column of the original matrix contains a significant fraction of zero-elements.

5.3 Application to Electronic Structure Methods

In Section 1, we motivated our method with the principal of linear scaling techniques in density functional theory, based on the nearsightedness of electronic matter. With respect to the matrices for which an inverse or an inverse p-th root needs to be calculated, this means that while for growing systems the total number of matrix elements increases quadratically, the density of the matrix decreases linearly with n^{-1}. Consequently, the number of nonzero elements in the matrix increases only linearly with n.

Based on this fact, the submatrix method is particularly suitable for solving these problems. In particular, since

$$n^{-1} < n^{-0.5}, \tag{16}$$

the density of matrices decreases faster than required in Equation (15). From that it follows that the asymptotic run time of the submatrix method in a single-threaded environment is limited by that of a precise inversion for the appliations discussed here.

Again, the strong advantage of the submatrix method is the possibility of parallel execution on many compute nodes. In the case of linear scaling methods in density functional theory, this means that for growing systems the execution can be parallelized onto more and more nodes while the size of the single submatrices stays constant. As long as the number of compute nodes can be scaled with n as well, the overall execution time can even be held constant.

6 PERFORMANCE EVALUATION

To evaluate the performance and scalability of the proposed method, we built a distributed implementation using MPI and OpenMP. We run this implementation on a compute cluster comprised of 65 nodes. Each node features two Intel Xeon E5-2670 CPUs with a total of 16 CPU cores and 64 GByte of memory. All nodes are connected via 40 Gbit/s QDR InfiniBand. We use one node as a control node, leaving the remaining 64 nodes with a total of 1024 CPU cores for handling the workload. In the following, we first describe details of our implementation, and then present results obtained from running our implementation on our compute cluster.

6.1 Details of our Implementation

Our implementation makes use of Intel MPI [14] to distribute work over a large number of compute nodes and to collect all results in order to build up the final result matrix. The MPI rank 0, in the following called main process, reads the input matrix stored in CSC format from persistent storage into memory. Metadata such as an

identifyer for the matrix, as well as its total size and number of nonzero elements, are then sent via `MPI_Bcast` to all nodes.

6.1.1 Data distribution and work assignment. There are different possible ways to make the input matrix available to all other MPI ranks, which we call worker processes in the following. In principal, it would be sufficient to send single submatrices to the workers which then perform the inversion. In this case, all submatrices would have to be constructed within the main process, which would clearly present a bottleneck. Instead, we make the whole input matrix available to all worker processes which then autonomously construct their submatrices. In our environment all systems have access to a shared file system which allows all processes to read in the input matrix from persistent storage. Since this scenario cannot generally be assumed, we additionally implemented distribution of data via `MPI_Bcast` to all worker processes. We found that, for the data we use in our evaluation, both variants provide comparable performance.

Assuming we have w workers, each worker needs to process $x = n/w$ submatrices. In our implementation, each worker processes a contiguous set of submatrices, i.e., the worker with rank k is responsible for submatrices $(k-1)x$ to $kx-1$. Therefore, depending on its rank and the total number of ranks, each worker process can determine autonomously, which of the submatrices it has to process.[3] It builds the submatrix according to Algorithm 1 and calls the LAPACK functions `dgetrf` to obtain an LU decomposition and `dgetri` to calculate the inverse of the submatrix. In our evaluation we use Intel MKL [13] as a highly optimized implementation for these LAPACK routines. After inversion, the worker selects the section of the result matrix which is relevant for the final result matrix and stores it in a buffer. Since the main process just needs to concatenate these buffers to create the final result matrix, a single call to `MPI_Gatherv` is sufficient to perform data collection and assembly after all submatrices have been processed.

6.1.2 Multi-threading using OpenMP. The described implementation already allows to distribute the load over many nodes and CPU cores. To utilize multiple CPU cores on a single node, multiple MPI ranks could be placed on a node, or multiple cores could be used for processing a single submatrix by using a multi-threaded LAPACK implementation. Having multiple MPI ranks on the same node comes at the cost of data duplication in memory and overall increased MPI communication load. Using multiple cores for a single submatrix operation has shown to provide lower speedup than additional parallelization of submatrix operations.

In our implementation, we therefore use OpenMP to process c submatrices in parallel on a node featuring c CPU cores. We do so by calling all submatrix operations within an OpenMP parallel for loop:

```
#pragma omp parallel for schedule(dynamic)
```

To allow OpenMP to fully utilize the available CPU cores, we explicitly disable the multi-threading functionality provided by Intel MKL by calling `mkl_set_num_threads(1)`.

6.1.3 Limitations of our Implementation. As described, each worker process is responsible for a contigous set of submatrices and all workers are responsible for the same number (±1) of submatrices. This can lead to workload imbalance between the different workers, if the input matrix exhibits a pattern such that certain sets of columns contain significantly more or significantly fewer nonzero values than other sets of columns. It is important to note that this is a limitation of our implementation and not a conceptual issue of the proposed submatrix method. In practice, there are different ways to deal with this issue in order to create an optimized implementation that does not exhibit this load imbalance:

Shuffling input matrices: To balance the load between all worker processes, the mapping between submatrices and workers can be shuffled randomly. Clusters of full colums which result in larger submatrices would then not be assigned to a single worker but distributed over all workers. This can, for example, be implemented using a pseudo-random but deterministic permutation, so that each worker can still autonomously determine the submatrices it is responsible for.

In our implementation, each worker concatenates the results of its submatrix operations in a buffer which is then sent as a whole to the main process. The main process therefore only needs to collect and concatenate w arrays for w worker processes. If submatrices are shuffled, collection and cocatenation of n arrays would be required in the main process instead. Apart from this, there is no additional computational effort required for this load balancing technique.

Dynamic work scheduling: Instead of assigning a fixed set of submatrices to a worker process, work can be scheduled dynamically. Each worker could request work packages from the main process using MPI. The size of these work packages can be chosen in the range from one single submatrix up to n/w submatrices in order to trade off load balancing and additional communication effort. Concepts similar to OpenMP's *guided* scheduling could also be implemented to minimize scheduling overhead. Note that in our implementation, we already use dynamic work scheduling for the parallel processing of multiple submatrices on a single node by using OpenMP's *dynamic* scheduler.

6.1.4 Availability. Our prototype implementation as well as scripts used in our evaluation are published under MIT license and can be found on https://github.com/pc2/SubmatrixMethod.

6.2 Results

6.2.1 Scalability for increasing number of CPU cores. We use our implementation of the submatrix method to calculate an approximate inverse of multiple random matrices with size $n = 32768$, condition number $\kappa = 2$ and density $d = 0.01$. We vary the number of utilized CPU cores in the range from 1 to 1024 and measure the total wall clock time required to obtain a result. We consider a set of balanced matrices whose columns have roughly the same number of nonzero elements[4] and a set of unbalanced matrices which exhibit visible patterns in the distribution of values[5]. Results are shown in Table 3 and Figure 5. As a reference, we also show the time required for a precise matrix inversion using Intel MKL's implementation of the `dgetrf` and `dgetri` routines, utilizing up to 16 CPU cores on a single node.

[3] Note that if the number of worker processes does not divide the size of the matrix, some workers need to process one additional submatrix.

[4] generated using `sprandsym(size,density,1/condition,kind)` with `kind=2`

[5] generated using `sprandsym(size,density,1/condition,kind)` with `kind=1`

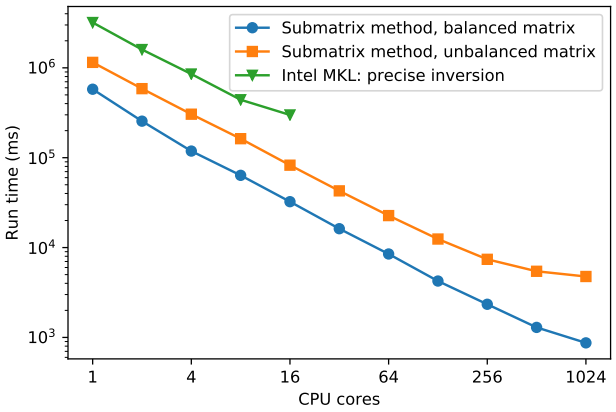

Figure 5: Scalability of the submatrix method for a random matrix of size $n = 32768$ and density $d = 0.01$.

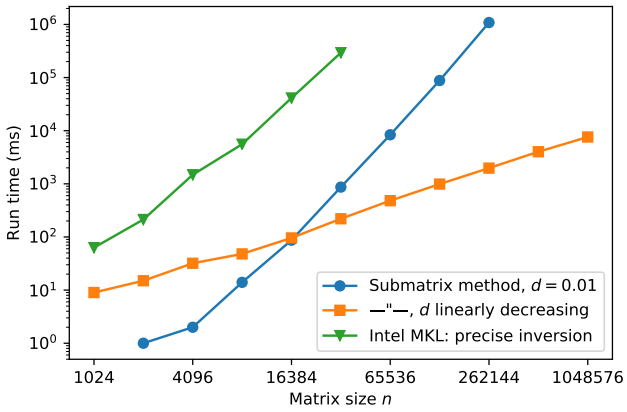

Figure 6: Required time for inversion of a matrix using the submatrix method (1024 cores) and Intel MKL (16 cores).

The results show that the submatrix method overall scales well over a large number of processors. Comparing the data for balanced and unbalanced matrices, two distinct effects can be observed:

(1) Even for a low number of CPU cores, the submatrix method performs better for balanced matrices. The reason for this is that if some columns contain significantly more nonzero elements than others, the resulting submatrices are larger in size and the time to process them increases cubically with their size.

(2) For the imbalanced matrices in our scenario, the curve starts to flatten at around 256 cores and scaling beyond 512 cores provides diminishing returns. The reason for this is that the number of submatrices per worker becomes small enough such that load imbalance between workers has an increasing effect. This effect could be countered by implementing some form of load balancing, as discussed in Section 6.1.3.

For over 512 cores, even for the balanced matrices in our scenario the additional speedup is limited. This is caused by the overall short runtime of the algorithm and therefore increased influence of communication time (around 32%).

On a single node, Intel MKL as well nearly scales linearly with the number of CPU cores. Only for 16 cores there is a slight efficiency drop, likely caused by the NUMA architecture of our compute nodes. Using ScaLAPACK to distribute execution of the utilized library functions over multiple nodes may allow to further increase the number of CPU cores. However, due to increasing communication overhead, the potential for scaling is limited in this case. Related work that uses ScaLAPACK for matrix inversion, describes decreasing performance for execution on more than 64 CPU cores [17].

Table 3: Time in ms required for inversion of a matrix with size $n = 32768$ and density $d = 0.01$ using the submatrix method.

	Balanced matrix		Unbalanced matrix	
Cores	**Wall time**	**Speedup**	**Wall time**	**Speedup**
1	578,140	1.0	1,150,366	1.0
2	255,081	2.3	586,778	2.0
4	118,534	4.9	304,941	3.8
8	63,644	9.1	162,792	7.1
16	32,405	17.8	82,571	13.9
32	16,216	35.7	42,760	26.9
64	8,485	68.1	22,692	50.7
128	4,242	136.3	12,447	92.4
256	2,339	247.2	7,402	155.4
512	1,293	447.1	5,447	211.2
1024	870	664.5	4,765	241.4

6.2.2 Run time for growing matrices. We now evaluate how the total execution time develops for increasing matrix sizes, given a fixed number of CPU cores. We consider two different scenarios: a fixed density of $d = 0.01$ and matrix sizes ranging from 2^{11} to 2^{18} and a density that decreases linearly with n as encountered in applications like electronic structure methods. For the latter, we set $d = 0.16 \cdot 1024/n$ and consider sizes from 2^{10} to 2^{20}.

The results of this evaluation are shown in Table 4 and Figure 6. In the table we also show the fraction of the total wall clock time spent on communication and the fraction of compute time spent on building the submatrices. Note that the assembly of the result matrix is performed implicitly by `MPI_Gatherv` and therefore accounted as communication time. It clearly shows that for matrices with linearly decreasing density, the required run time only increases linearly with the matrix size, as expected based on the discussion in Section 5.3. Combining this result with the possibility for linear performance scaling with the number of CPU cores, the run time can be held constant by increasing the number of cores with n for growing matrices. The data also shows that for increasing size of the submatrices, as shown in the upper half of Table 4, the overhead required for communication and for building the submatrices decreases. For fixed-size submatrices, the overhead stays relatively constant. Note that times for communication fluctuate in our measurements due to shared usage of the underlying InfiniBand network.

7 CONCLUSION

In this work we presented the submatrix method, which can be used to calculate an approximate inverse of matrices, as well as inverse p-th roots. Following the idea of Approximate Computing, it allows the result to deviate from an exactly calculated solution in order to utilize the sparsity of the input matrix and to allow massively parallel execution of the involved calculations. For an $n \times n$ matrix, the workload can be distributed over n nodes.

A particularly well suited application for the submatrix method are electronic structure methods in density functional theory. In these applications, for growing matrices their density decreases linearly at the same time. In this case, the submatrix method exhibits a linear increase in execution time for growing systems. As long as the number of available CPU cores can be scaled with the same rate, execution time can even be held constant.

We showed that the error introduced by using the submatrix method is limited for well-conditioned input matrices and demonstrated its use for preconditioning of ill-conditioned matrices. We discussed the scalability of the algorithm both theoretically and in a practical evaluation on a large compute cluster.

An emerging question for future work is how this method can be combined with iterative methods to refine the solution or to increase efficiency of the submatrix operations. Additionally, the dense nature of the submatrices make the submatrix operations well suited for the use of accelerator hardware which may be explored in future to develop high-performance implementations of the submatrix operations.

Table 4: Time in ms required for inversion of a matrix using the submatrix method on 1024 cores.

Size	Density	Wall time	MPI Comm.	Submat. Constr.
2,048	1.0×10^{-2}	1	—	—
4,096	1.0×10^{-2}	2	—	—
8,192	1.0×10^{-2}	14	57.1%	58.8%
16,384	1.0×10^{-2}	87	39.1%	60.5%
32,768	1.0×10^{-2}	868	31.7%	57.8%
65,536	1.0×10^{-2}	8,380	13.5%	56.8%
131,072	1.0×10^{-2}	87,977	5.0%	47.0%
262,144	1.0×10^{-2}	1,085,176	1.5%	36.8%
1,024	1.6×10^{-1}	9	22.2%	61.3%
2,048	8.0×10^{-2}	15	26.7%	61.0%
4,096	4.0×10^{-2}	32	37.5%	60.5%
8,192	2.0×10^{-2}	48	33.3%	60.3%
16,384	1.0×10^{-2}	96	38.5%	60.4%
32,768	5.0×10^{-3}	220	51.7%	60.9%
65,536	2.5×10^{-3}	482	54.4%	61.4%
131,072	1.3×10^{-3}	990	54.7%	62.2%
262,144	6.3×10^{-4}	1977	55.2%	63.0%
524,288	3.1×10^{-4}	4020	56.8%	63.7%
1,048,576	1.6×10^{-4}	7609	49.9%	64.0%

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 716142) and from the German Research Foundation (DFG) under the project PerfinienCC (grant agreement No PL 595/2-1). Compute resources were provided by the Paderborn Center for Parallel Computing (PC^2).

REFERENCES

[1] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: A portable linear algebra library for high-performance computers. In *Proc. SUPERCOMPUTING '90*. 2–11.

[2] W. E. Arnoldi. 1951. The principle of minimized iteration in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.* 9 (1951), 17–29.

[3] Michael W Berry, Dani Mezher, Bernard Philippe, and Ahmed Sameh. 2006. Parallel algorithms for the singular value decomposition. *Statistics Textbooks and Monographs* 184, 117 (2006), 31.

[4] D. A. Bini, N. J. Higham, and B. Meini. 2005. Algorithms for the matrix pth root. *Numerical Algorithms* 39, 4 (2005), 349–378.

[5] Laura Susan Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. 1996. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In *Proc. 1996 ACM/IEEE Conf. on Supercomputing*. IEEE Computer Society, Washington, DC, USA, Article 5.

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3 ed.). MIT Press.

[7] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (Jan 1998), 46–55.

[8] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1, Article 1 (Dec. 2011), 25 pages.

[9] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. 2011. High Performance Matrix Inversion Based on LU Factorization for Multicore Architectures. In *Proc. 2011 ACM Int. Workshop on Many Task Computing on Grids and Supercomputers (MTAGS '11)*. ACM, New York, NY, USA, 33–42.

[10] Stefan Goedecker. 1999. Linear scaling electronic structure methods. *Reviews of Modern Physics* 71 (Jul 1999), 1085–1123. Issue 4.

[11] Nicholas J Higham and Lijing Lin. 2011. A Schur–Padé algorithm for fractional powers of a matrix. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 1056–1078.

[12] Nicholas J. Higham and Lijing Lin. 2013. An Improved Schur–Padé Algorithm for Fractional Powers of a Matrix and Their Fréchet Derivatives. *SIAM J. Matrix Anal. Appl.* 34, 3 (2013), 1341–1360.

[13] Intel. 2017. Math Kernel Library. http://www.intel.com/software/products/mkl/

[14] Intel. 2017. MPI Library. https://software.intel.com/en-us/intel-mpi-library

[15] P. Klavík, A. C. I. Malossi, C. Bekas, and A. Curioni. 2014. Changing Computing Paradigms Towards Power Efficiency. *Philosophical Transactions of the Royal Society A: Mathematical, Physical & Engineering Sciences* 372, 2018 (2014).

[16] W. Kohn. 1999. Nobel Lecture: Electronic structure of matter – wave functions and density functionals. *Reviews of Modern Physics* 71 (Oct 1999), 1253–1266. Issue 5.

[17] Mariana Kolberg, Gerd Bohlender, and Dalcidio Claudio. 2008. Improving the Performance of a Verified Linear System Solver Using Optimized Libraries and Parallel Computation. In *Proc. 8th Int. Conf. High Performance Computing for Computational Science (VECPAR)*, José M. Laginha M. Palma, Patrick R. Amestoy, Michel Daydé, Marta Mattoso, and João Correia Lopes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 13–26.

[18] C. Lanczos. 1950. An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators. *J. Res. Nat. Bur. Standards* 45, 4 (Oct. 1950), 255–282.

[19] Michael Lass, Thomas D. Kühne, and Christian Plessl. 2017. Using Approximate Computing for the Calculation of Inverse Matrix p-th Roots. *IEEE Embedded Systems Letters* (2017). arXiv:1703.02283 Accepted for publication.

[20] François Le Gall. 2014. Powers of Tensors and Fast Matrix Multiplication. In *Proc. 39th Int. Symp. on Symbolic and Algebraic Computation (ISSAC '14)*. ACM, New York, NY, USA, 296–303.

[21] R.B. Lehoucq, D.C. Sorensen, and C. Yang. 1998. *ARPACK Users' Guide: Solution of Large-scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. Society for Industrial and Applied Mathematics.

[22] Lin Lin, Chao Yang, Jianfeng Lu, Lexing Ying, and Weinan E. 2011. A Fast Parallel Algorithm for Selected Inversion of Structured Sparse Matrices with Application to 2D Electronic Structure Calculations. *SIAM Journal on Scientific Computing* 33, 3 (June 2011), 1329–1351.

[23] Kristyn J Maschhoff and Danny C Sorensen. 1996. P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures. In *Int. Workshop on Applied Parallel Computing*. Springer, 478–486.

[24] Message Passing Interface Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. University of Tennessee, Knoxville, TN, USA.

[25] Stephan Mohr, William Dawson, Michael Wagner, Damien Caliste, Takahito Nakajima, and Luigi Genovese. 2017. Efficient Computation of Sparse Matrix Functions for Large-Scale Electronic Structure Calculations: The CheSS Library. *Journal of Chemical Theory and Computation* 13, 10 (2017), 4684–4698. PMID: 28873312.

[26] S. Mohr, L.E. Ratcliff, P. Boulanger, L. Genovese, D. Caliste, T. Deutsch, and S. Goedecker. 2014. Daubechies wavelets for linear scaling density functional theory. *J Chem Phys* 140, 20 (2014), 204110.

[27] E. Prodan and W. Kohn. 2005. Nearsightedness of electronic matter. *Proc. of the National academy of Sciences of the United States of America* 102, 33 (2005), 11635–11638.

[28] D. Richters and T. D. Kühne. 2014. Self-consistent field theory based molecular dynamics with linear system-size scaling. *Journal of Chemical Physics* 140, 13 (2014), 134109.

[29] Dorothee Richters, Michael Lass, Andrea Walther, Christian Plessl, and Thomas D. Kühne. 2018. A General Algorithm to Calculate the Inverse Principal p-th Root of Symmetric Positive Definite Matrices. *Communications in Computational Physics* (2018). arXiv:1703.02456 Accepted for publication.

[30] G. Schulz. 1933. Iterative Berechung der reziproken Matrix. *ZAMM - Zeitschrift fur Angewandte Mathematik und Mechanik* (1933).

[31] Kai Shen. 2006. Parallel Sparse LU Factorization on Different Message Passing Platforms. *J. Parallel and Distrib. Comput.* 66, 11 (Nov. 2006), 1387–1403.

[32] A. Frank van der Stappen, Rob H. Bisseling, and Johannes G.G. van de Vorst. 1993. Parallel Sparse LU Decomposition on a Mesh Network of Transputers. *SIAM J. Matrix Anal. Appl.* 14, 3 (1993), 853–879.